Laboratorium 3.

0) Przygotowanie środowiska

W konsoli/terminalu wpisujemy kolejno

```
$ cd
$ mkdir haskell-lab3
$ cd haskell-lab3
```

1) Funkcje anonimowe i *currying*

1. W konsoli GHCi wpisujemy kolejno

```
ghci> :t (\x -> \y -> x + y)
ghci> :t (\x y -> x + y) -- porównujemy wynik z poprzednim

ghci> :t (\x -> \y -> x + y) 1
ghci> :t (\x y -> x + y) 1 -- porównujemy wynik z poprzednim

ghci> :t (\x -> \y -> x + y) 1 2
ghci> :t (\x y -> x + y) 1 2 -- porównujemy wynik z poprzednim

ghci> (\x -> \y -> x + y) 1 2
ghci> (\x y -> x + y) 1 2 -- porównujemy wynik z poprzednim

ghci> let f1 = \x -> \y -> x + y
ghci> :t f1
ghci> f1 1 2

ghci> let f2 = \x y -> x + y
ghci> :t f2 -- porównujemy z wynikiem dla f1
ghci> f2 1 2 -- porównujemy z wynikiem dla f1

ghci> let f3 x = \y -> x + y
ghci> :t f3 -- porównujemy z wynikami dla f1 i f2
ghci> f3 1 2 -- porównujemy z wynikami dla f1 i f2

ghci> let f4 x y = x + y
ghci> :t f4 -- porównujemy z wynikami dla f1, f2 i f3
ghci> f4 1 2 -- porównujemy z wynikami dla f1, f2 i f3

ghci> let f5 = \(x,y) -> x + y
ghci> :t f5 -- porównujemy z wynikami dla f1, ... , f4
ghci> f5 1
ghci> f5 (1,2) -- porównujemy z wynikami dla f1, ... , f4

ghci> let f6 (x,y) = x + y
```

```
ghci> :t f6 -- porównujemy z wynikami dla f1, ... , f5
ghci> f6 1
ghci> f6 (1,2) -- porównujemy z wynikami dla f1, ... , f5

ghci> :t (\x y -> x + y) :: Int -> Int -> Int
ghci> :t (\x y -> x + y)
ghci> ((\x y -> x + y) :: Int -> Int -> Int) 1 2
```

2. ***Zadania***:

    1. Napisać funkcje anonimowe (wyrażenia lambda) odpowiadające:

        - $f_1(x) = x - 2;\ x \in \mathbb{R}$
        - $f_2(x, y) = \sqrt{(x^2 + y^2)};\ x, y \in \mathbb{R}$
        - $f_3(x, y, z) = \sqrt{(x^2 + y^2 + z^2)};\ x, y, z \in \mathbb{Z}$

        Uwaga: Sprawdzić działanie dla wybranych wartości argumentów

    2. Napisać funkcje anonimowe (wyrażenia lambda) odpowiadające:

        ```
        (2*), (*2), (2^), (^2), (2/), (/3), (4-)
        ```

        Uwaga: jw.

    3. (*opcjonalne*) Napisać funkcje anonimowe (wyrażenia lambda) odpowiadające:

        ```
        sqrt, abs, log, id, const
        ```

        Uwaga: jw.

    4. Npisać funkcje anonimowe (wyrażenia lambda) odpowiadające:

        ```
        f7 x = if x `mod` 2 == 0 then True else False

        f8 x = let y = sqrt x in 2 * y^3 * (y + 1)

        f9 1 = 3
        f9 _ = 0
        ```

        Uwaga: jw.

2) Funkcje wyższego rzędu: funkcje jako parametry/argumenty

1. W pliku ex2.hs wpisujemy

```
sum' :: Num a => [a] -> a
sum' []     = 0
sum' (x:xs) = x + sum' xs
```

zapisujemy zmiany i sprawdzamy działanie funkcji

2. ***Zadania***:
    1. Na podstawie powyższej definicji napisać funkcję `sumSqr'` (sumującą kwadraty elementów listy)
    2. Zdefiniować funkcję

    ```
    sumWith :: Num a => (a -> a) -> [a] -> a
    sumWith f ...
    ```

    uogólniającą `sum` i `sumSqr'`
    3. Wykorzystując funkcję `sumWith` zdefiniować

    ```
    sum    = sumWith ...
    sumSqr  = sumWith ...
    sumCube = sumWith ...
    sumAbs  = sumWith ...
    ```

    Sprawdzić działanie powyższych funkcji, dla wybranych danych wejściowych, np.

    ```
    sumSqr [1..10]
    ```

    4. Wykorzystując `sumWith` (bez definiowania nowej funkcji) obliczyć w GHCi

    $\sum_{i=1}^{15} i^5$

    5. Wykorzystując funkcję `sumWidth` zdefiniować

    ```
    listLength = sumWith ...
    ```

    6. (*opcjonalne*) Na podstawie (schematu) definicji `sum'` napisać funkcję

    ```
    prod' :: Num a => [a] -> a
    ```

    obliczającą iloczyn elementów listy
    7. (*opcjonalne*) Na podstawie definicji `sumWith` napisać funkcję

    ```
    prodWith :: Num a => (a -> a) -> [a] -> a
    ```

    a następnie funkcje

    ```
    prod    = prodWith ...
    prodSqr  = prodWith ...
    prodCube = prodWith ...
    prodAbs  = prodWith ...
    ```

    8. (*opcjonalne*) W jaki sposób można uogólnić funkcje `sumWith` i `prodWith` ?
    9. (*opcjonalne*) Zmodyfikować deklaracje

```
sumWith :: Num a => (a -> a) -> [a] -> a
prodWith :: Num a => (a -> a) -> [a] -> a
```

tak, aby można było (na ich podstawie) zdefiniować funkcje

```
sumSqrt  = sumWith ... -- suma pierwiastków (kwadratowych) elementów listy
prodSqrt = prodWith ... -- iloczyn pierwiastków (kwadratowych) elementów listy
```

3) Funkcje wyższego rzędu: funkcje jako wyniki

1. W pliku ex3.hs wpisujemy

```
sqr x = x^2

funcFactory n = case n of
 1 -> id
 2 -> sqr
 3 -> (^3)
 4 -> \x -> x^4
 5 -> intFunc
 _ -> const n
 where
   intFunc x = x^5
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie funkcji `funcFactory`, np.

```
ghci> let cub = funcFactory 3
ghci> cub 4
```

Sprawdzamy typy `funcFactory` i `funcFactory 3`

2. ***Zadania***:

   1. Napisać funkcję

   ```
   expApproxUpTo :: Int -> Double -> Double
   expApproxUpTo n = ...
   ```

   zwracającą rozwinięcie funkcji e^x w szereg MacLaurina o długości `n+1, n < 6`, tzn. `expApproxUpTo n` $= \sum_{k=0}^{n} \frac{x^k}{k!}$.
   Czy da się to rozwiązanie uogólnić tak, aby funkcja `expApproxUpTo` zwracała rozwinięcia dla dowolnego `n`?

   2. (*opcjonalne*) Napisać funkcję

   ```
   dfr :: (Double -> Double) -> Double -> (Double -> Double)
   dfr f h = ...
   ```

zwracającą dla zadanej funkcji `f` przybliżenie jej pochodnej obliczone wg schematu różnicowego $f'(x_0, h) \approx \frac{f(x_0+h)-f(x_0)}{h}$.

Sprawdzić dokładność uzyskiwanych wyników w zależności od wartości `h`.

3. (*opcjonalne*) Napisać funkcję

```
dfc :: (Double -> Double) -> Double -> (Double -> Double)
dfc f h = ...
```

zwracającą dla zadanej funkcji `f` przybliżenie jej pochodnej obliczone wg schematu różnicowego $f'(x_0, h) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$.

Sprawdzić dokładność uzyskiwanych wyników w zależności od wartości `h` i porównać z poprzednimi.

4. (*opcjonalne*) Napisać funkcję

```
d2f :: (Double -> Double) -> Double -> (Double -> Double)
d2f f h = ...
```

obliczającą przybliżenie drugiej pochodnej funkcji `f`

4) Funkcje jako elementy struktur danych

1. W pliku ex4.hs wpisujemy

```
funcList :: [ Double -> Double ]
funcList = [ \x -> (sin x)/x, \x -> log x + sqrt x + 1, \x -> (exp 1) ** x ]

evalFuncListAt :: a -> [a -> b] -> [b]
evalFuncListAt x [] = []
evalFuncListAt x (f:fs) = f x : evalFuncListAt x fs
```

zapisujemy zmiany, wczytujemy plik do GHCi i wpisujemy w konsoli

```
ghci> evalFuncListAt 1 funcList
ghci> evalFuncListAt (-3) [ id, abs, const 5, \y -> 2 * y + 8 ]
```

2. W pliku ex4.hs dodajemy

```
displEqs :: (Double -> Double, Double -> Double)
displEqs = (\t -> 4 * t^2 + 2 * t, \t -> 3 * t^2)
```

zapisujemy zmiany, wczytujemy plik do GHCi i wpisujemy w konsoli

```
ghci> let (x_t, y_t) = (fst displEqs, snd displEqs)
```

```
ghci> x_t 1
ghci> y_t 1
```

3. ***Zadania***:
    1. (*opcjonalne*) W GHCi utworzyć nową listę funkcji `funcListExt` poprzez dodanie do `funcList` funkcji $f(x) = \sqrt{(1+x)}$; sprawdzić rozmiar `funcListExt` i wywołać `evalFuncListAt` (dla wybranego punktu i listy `funcListExt`)
    2. (*opcjonalne*) Wykorzystując zdefiniowane wcześniej funkcje `dfc` i `d2f` w konsoli GHCi obliczyć wektory reprezentujące prędkości (`velocEqs`) i przyspieszenia (`accelEqs`)

5) Operator złożenia funkcji $(.)$ (i notacja *point-free*)

1. W konsoli GHCi wpisujemy kolejno następujące wyrażenia i analizujemy wyniki

```
ghci> :t (.)
ghci> :i (.) --  co oznacza "infixr 9"?

ghci> let f = (+1) -- vs. let f x = x + 1 vs. let f x = (+1) x
ghci> let g = (*2) -- vs. let g x = x * 2 vs. let g x = (*2) x
ghci> let h = (^3) -- vs. let h x = x ^ 3 vs. let h x = (^3) x

ghci> let fg = f . g      -- vs. let fg x  = (f . g) x
ghci> let gh = g . h      -- vs. let gh x  = (g . h) x
ghci> let fgh = f . g . h -- vs. let fgh x = (f . g . h) x

ghci> fgh 3
ghci> f . g . h 3
ghci> (f . g . h) 3
ghci> (f . g . h) 3 == f (g (h (3)))

ghci> (f . id) 3 == (id . f) 3
ghci> (f . id) 3 == f 3

ghci> (f . gh) 3 == (fg . h) 3
ghci> (f . (g . h)) 3 == ((f . g) . h) 3
ghci> fgh 3 == (f . gh) 3
ghci> (f . g) 3 == (.) f g 3

ghci> let u2 x y = x^2 + y^2

ghci> (f . (u2 4) . g) 3
ghci> (f . u2 4 . g) 3
ghci> :t u2 4

ghci> ((u2 1) . (u2 2) . (u2 3)) 1
ghci> (u2 1 . u2 2 . u2 3) 1
```

```
ghci> ((+) 2 . g . h) 3
ghci> ((2+) . g . h) 3
```

2. ***Zadania***:
    1. Wykorzystując funkcje `sort` i `reverse` oraz operator `.` napisać funkcję sortującą malejąco podaną listę

        ```
        sortDesc :: Ord a => [a] -> [a]
        sortDesc xs = ...
        ```

        Uwaga: potrzebny będzie `import Data.List`
    2. Przepisać funkcję `sortDesc` do wersji *point-free*
    3. W GHCi zdefiniować funkcję

        ```
        let w3 = \x y z -> sqrt (x^2 + y^2 + z^2)
        ```

        a następnie uzupełnić poniższe wyrażenie

        ```
        (f . w3 ___ . h) 3
        ```

    4. (*opcjonalne*) Napisać funkcję

        ```
        are2FunsEqAt :: Eq a => (t -> a) -> (t -> a) -> [t] -> Bool
        are2FunsEqAt f g xs = ... -- are2FunsEqAt (+2) (\x -> x + 2) [1..1000] = True
        ```

        sprawdzającą, czy funkcje `f` i `g` mają równe wartości we wszystkich punktach podanych w liście `xs`
    5. (*opcjonalne*) Napisać funkcję

        ```
        infixl 9 >.>
        (>.>) :: (a -> b) -> (b -> c) -> (a -> c)
        g >.> f = ...
        ```

        i sprawdzić jej działanie
    6. (*opcjonalne*) Napisać funkcję (kompozycja/złożenie listy funkcji)

        ```
        composeFunList :: [a -> a] -> (a -> a)
        composeFunList ...
        ```

    7. (*opcjonalne*) Przeanalizować typ operatora `(.).(.)`, a następnie zastosować go do złożenia wybranych funkcji (odpowiedniego typu)

6) Operator “aplikacji” funkcji ($)

1. W konsoli GHCi wpisujemy kolejno następujące wyrażenia i analizujemy wyniki

```
ghci> :t ($)
ghci> :i ($) -- porównujemy z wynikiem ":i (.)"
ghci> :t ($!)
ghci> :i ($!)

ghci> ($) f 3 == f $ 3
ghci> (($) f 3) == (f $ 3)
ghci> ($) f 3 == (f $ 3)
ghci> f (g (h (3)))
ghci> f (g (h $ 3))
ghci> f (g $ h $ 3)
ghci> f $ g $ h $ 3

ghci> (f $ g $ h $ 3) == (f $ (g $ (h $ 3)))
ghci> f $ g $ h $ 3 == (f $ (g $ (h $ 3)))
ghci> (f $ g $ h $ 3) == f $ (g $ (h $ 3))
ghci> :i (==) -- porównujemy z wynikiem ":i ($)"

ghci> _3 = ($3)
ghci> f 3
ghci> _3 f -- :)
ghci> f 3 == _3 f

ghci> ((,) $ 1) 2
```

2. ***Zadania***:
    1. Dodać nawiasy w wyrażeniu `(,) $ 1 $ 2` , aby otrzymać wynik `(1,2)`
    2. Sprawdzić (eksperymentalnie) równoważność poniższych wyrażeń

        ```
        f $ g $ h 3
        (f . g . h) 3
        f . g . h $ 3
        ```

        Które z nich jest najbardziej czytelne?
    3. (*opcjonalne*) Dodać nawiasy w wyrażeniu `f . $ g . h 3` , aby otrzymać wynik `55`

7) Funkcje wyższego rzędu: `filter`

1. W pliku `ex7.hs` wpisujemy funkcję

```
onlyEven [] = []
onlyEven (x:xs)
 | x `mod` 2 == 0 = x : onlyEven xs
 | otherwise      = onlyEven xs
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy jej działanie

2. ***Zadania***:

    1. (*opcjonalne*) Napisać (wg podanego schematu) definicje funkcji `onlyOdd` i `enableUpper`

```
ghci> onlyOdd [1..10] -- [1,3,5,7,9]
ghci> onlyUpper "My name is Inigo Montoya. You killed my father. Prepare to die
```

2. Uogólnić poprzednie rozwiązania wprowadzając funkcję `filter'`

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p ...

onlyEven  = filter' ...
onlyOdd   = filter' ...
onlyUpper = filter' ...
```

3. Porównać czasy wykonania (opcja `:set +s`)

```
ghci> length (onlyEven [1..10^6])
ghci> length (filter even [1..10^6]) -- filter z biblioteki standardowej
```

4. Zmodyfikować powyższe wyrażenia tak, aby nie zawierały nawiasów (użyć `.` i/lub `$`)
5. Przepisać używając *list comprehensions*

```
length (filter even [1..10^6])
```

Zmierzyć czas wykonania i porównać go z uzyskanym w poprzednim zadaniu

6. Obliczyć w GHCi i przeanalizować poniższe wyrażenia

```
ghci> filter (\s -> length s == 2) ["a", "aa", "aaa", "b", "bb"]
ghci> filter (\(x,y) -> x > y) [(1,2), (2,2), (2,1), (2,2), (3,2)]
ghci> filter (\xs -> sum xs > 300) [[1..5], [56..60], [101..105]]
ghci> length . filter (\f -> f 2 > 10) $ [(+5), (*5), (^5), \x -> 3 * x + 7]
```

8) Funkcje wyższego rzędu: `map`

1. W pliku `ex8.hs` wpisujemy funkcję

```
doubleElems []     = []
doubleElems (x:xs) = 2 * x : doubleElems xs
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy jej działanie

2. ***Zadania***:

1. (*opcjonalne*) Napisać (wg podanego schematu) definicje funkcji `sqrElems` i `lowerCase`

```
ghci> sqrElems [1..3] -- [1,4,9]
ghci> lowerCase "ABCD" -- "abcd", konieczny import Data.Char
```

2. Uogólnić poprzednie rozwiązania wprowadzając funkcję `map'`

```
map' :: (a -> b) -> [a] -> [b]
map' f ...

doubleElems = map' ...
sqrElems    = map' ...
lowerCase   = map' ...
```

3. Przepisać powyższe funkcje używając *list comprehensions*
4. Porównać czasy wykonania (opcja `:set +s`)

```
ghci> length . filter even $ doubleElems [1..10^7]
ghci> length . filter even . map (*2) $ [1..10^7] -- map z biblioteki standardo
```

5. (*opcjonalne*) Przepisać powyższe wyrażenia używając *list comprehensions*, a następnie zmierzyć czasy wykonania i porównać je z uzyskanymi w poprzednim zadaniu
6. Obliczyć w GHCi i przeanalizować poniższe wyrażenia

```
ghci> map (*2) [1..10]
ghci> map (^2) [1..10]
ghci> map toLower "ABCD" -- konieczny import Data.Char
ghci> length . filter (>10) . map ($ 2) $ [(+5), (*5), (^5), \x -> 3 * x + 7]
ghci> map show [1..10]
ghci> map length [[1],[1,2],[1,2,3]]
ghci> map (map length) [ [[1],[1,2],[1,2,3]], [[1],[1,2]] ]
ghci> map (\(x,y) -> (y,x)) [(1,'a'), (2,'b'), (3,'c')]
ghci> map (\(x,y) -> y) [(1,'a'), (2,'b'), (3,'c')]
ghci> map (\s -> (s, length s)) ["manuscripts","do","not","burn"]
```

7. (*opcjonalne*) Zdefiniować funkcję `evalFuncListAt` wykorzystując `map`

```
evalFuncListAt :: a -> [a -> b] -> [b]
evalFuncListAt x = map ...
```

8. (*opcjonalne*) Napisać *list comprehensions* odpowiadające powyższym wyrażeniom
9. (*opcjonalne*) Przeanalizować z punktu widzenia złożoności obliczeniowej (czasowej i pamięciowej) dwa warianty połączenia operatorów `map` i `filter`

```
map f . filter p $ xs
filter p . map f $ xs
```

9) Funkcje wyższego rzędu: `foldr` i `foldl`

1. W pliku `ex9.hs` wpisujemy funkcje

```
sumWith g []     = 0
sumWith g (x:xs) = g x + sumWith g xs -- (+) (g x) (sumWith g xs)
```

```
prodWith g []     = 1
prodWith g (x:xs) = g x * prodWith g xs -- (*) (g x) (prodWith g xs)
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy ich działanie

2. W pliku `ex9.hs` wpisujemy funkcje

```
sumWith' :: Num a => (a -> a) -> [a] -> a
sumWith' = go 0
 where
   go acc g [] = acc
   go acc g (x:xs) = go (g x + acc) g xs

prodWith' :: Num a => (a -> a) -> [a] -> a
prodWith' = go 1
 where
   go acc g [] = acc
   go acc g (x:xs) = go (g x * acc) g xs
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy ich działanie

3. ***Zadania***:

    1. Uogólnić `sumWith` i `prodWith` wprowadzając funkcję `foldr'`

    ```
    foldr' :: (a -> b -> b) -> b -> [a] -> b
    foldr' f z ...

    sumWith'' g  = foldr' (\x acc -> g x + acc) 0
    prodWith'' g = foldr' ...
    ```

    Sprawdzić działanie nowych wersji `sumWith` i `prodWith`

    2. Uogólnić `sumWith'` i `prodWith'` wprowadzając funkcję `foldl'`

    ```
    foldl' :: (b -> a -> b) -> b -> [a] -> b
    foldl' f z ...

    sumWith''' g  = foldl' (\acc x -> g x + acc) 0
    prodWith''' g = foldl' ...
    ```

    Sprawdzić działanie nowych wersji `sumWith'` i `prodWith'`

    3. (*opcjonalne*) Porównać czasy wykonania (opcja `:set +s`)

    ```
    ghci> foldr' (+) 0 [1..10^6]
    ghci> foldr (+) 0 [1..10^6] -- foldr z biblioteki standardowej
    ```

4. (*opcjonalne*) Porównać czasy wykonania (opcja `:set +s`)

```
ghci> foldr (\x acc -> x + 1 + acc) 0 [1..10^6]
ghci> foldl (\acc x -> x + 1 + acc) 0 [1..10^6]
ghci> foldl' (\acc x -> x + 1 + acc) 0 [1..10^6] -- konieczny import Data.List
ghci> sum . map (+1) $ [1..10^6]
ghci> sum [x + 1 | x <- [1..10^6]]
```

5. Obliczyć w GHCi i przeanalizować poniższe wyrażenia

```
ghci> let strList1 = ["My", "name", "is", "Inigo", "Montoya"]
ghci> foldr (++) [] strList1
ghci> foldr (\x acc -> x ++ " " ++ acc) [] strList1
ghci> foldr1 (\x acc -> x ++ " " ++ acc) strList1

ghci> foldr (\_ acc -> 1 + acc) 0 list1To5

ghci> let list1To5 = [1..5]
ghci> foldr (:) [] list1To5
ghci> foldl (:) [] list1To5
ghci> foldl (\acc x -> x : acc) [] list1To5
ghci> foldr (\x xs -> xs ++ [x]) [] list1To5

ghci> let listRand = [1,4,2,6,5,3]
ghci> foldr1 max listRand
ghci> foldr1 min listRand

ghci> let listBool = [True, False, True, False]
ghci> foldr (||) False listBool
ghci> foldr (&&) True listBool

ghci> foldr (+) 0 list1To5 == foldl (+) 0 list1To5
ghci> foldr (*) 0 list1To5 == foldl (*) 0 list1To5
ghci> foldr (-) 0 list1To5 == foldl (-) 0 list1To5

ghci> let list321 = [3,2,1]
ghci> foldr (-) 0 list321
ghci> foldr1 (-) list321
ghci> foldl (-) 0 list321
ghci> foldl1 (-) 0 list321
```

6. Obliczyć w GHCi i przeanalizować wyrażenia

```
ghci> foldr (\x acc -> "(" ++ x ++ " f " ++ acc ++ ")") "z" ["1","2","3"]
ghci> foldr1 (\x acc -> "(" ++ x ++ " f " ++ acc ++ ")") ["1","2","3"]
```

```
ghci> foldl (\acc x -> "(" ++ acc ++ " f " ++ x ++ ")") "z" ["1","2","3"]
ghci> foldl1 (\acc x -> "(" ++ acc ++ " f " ++ x ++ ")") ["1","2","3"]
```

7. (*opcjonalne*) Napisać definicje funkcji:
   - `map` wykorzystując `foldr`
   - `map` wykorzystując `foldl`
   - `filter` wykorzystując `foldr`
   - `filter` wykorzystując `foldl`
   - `foldl` wykorzystując `foldr`
   - `foldr` wykorzystując `foldl`

10) Funkcje: `zip` , `unzip` i `zipWith`

1. W konsoli GHCi wpisujemy kolejno

```
ghci> zip [1,2,3] ['a','b']
ghci> unzip [(1,'a'),(2,'b')]
ghci> unzip (zip [1,2,3] ['a','b'])
ghci> zip [1,2] [10,20] == zipWith (,) [1,2] [10,20]

ghci> let endlessList = [1..]
ghci> take 5 (zip endlessList (tail endlessList))
```

   i analizujemy wyniki

2. ***Zadania***:

   1. Wykorzystując `zip` lub `zipWith` napisać funkcję

```
isSortedAsc :: Ord a => [a] -> Bool
isSortedAsc xs ... -- isSortedAsc [1,2,2,3] -> True, isSortedAsc [1,2,1] -> Fal
```

   2. Napisać funkcję

```
everySecond :: [t] -> [t]
everySecond xs = ... -- everySecond [1..8] -> [1,3,5,7]
```

   3. (*opcjonalne*) Napisać funkcje

```
zip3' :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3' ...

unzip3' :: [(a, b, c)] -> ([a], [b], [c])
unzip3' ...

isSortedDesc :: Ord a => [a] -> Bool
isSortedDesc xs ... -- isSortedDesc [3,2,2,1] -> True, isSortedDesc [1,2,3] ->
```

```
isSorted :: Ord a => [a] -> Bool
isSorted xs ... -- isSorted [1,2,2,3] -> True, isSorted [3,2,1] -> True
```

4. (*opcjonalne*) Przeanalizować następującą definicję ciągu Fibonacciego

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs) :: [Int]
```

Uwaga: Pomocne może być porównanie z definicjami

```
ones = 1 : ones
nats = 1 : map (+1) nats
```

11) Funkcje `concat` i `concatMap`

1. W pliku `ex11.hs` definiujemy funkcję

```
concat' :: [[a]] -> [a]
concat' []     = []
concat' (x:xs) = x ++ concat' xs
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy jej działanie

```
ghci> concat' ["abc", "def"]
ghci> concat' [[1,2],[3,4]]
ghci> (concat' . concat') [ [[1,2], [3,4]] , [[5,6], [7,8]] ]
```

2. W konsoli GHCi sprawdzamy typ funkcji bibliotecznej `concat` i sprawdzamy jej działanie, np.

```
ghci> concat' ["abc", "def"] == concat ["abc", "def"]
```

3. W konsoli GHCi sprawdzamy typ funkcji bibliotecznej `concatMap`. Jaki jest związek funkcji `concatMap` z funkcjami `map` i `concat`? Czy jest to zwykłe złożenie?
4. ***Zadania***:
    1. (*opcjonalne*) Napisać definicję funkcji `concat` wykorzystując
        - *list comprehension*
        - `foldr`
    2. (*opcjonalne*) Uzupełnić poniższe wyrażenia

```
concat ___ map (___) ___ [1..5] -- [2,4,6,8,10]
concatMap (___) [1..5] -- [2,4,6,8,10]
concatMap (___) ["Ready", "Steady", "Go"] -- "Ready!Steady!Go!"
```

12) Wzorzec *Collection pipeline*

1. W pliku `ex12.hs` wpisujemy

```
import Data.Char
import Data.List

capitalize :: [Char] -> [Char]
capitalize [] = []
capitalize (x:xs) = toUpper x : (map toLower xs)

formatStr s = foldr1 (\w s -> w ++ " " ++ s) .
          map capitalize .
          filter (\x -> length x > 1) $
          words s
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie

```
ghci> formatStr "tomasz  t  ,     bogdan anna . Jerzy j    maria"
```

2. W pliku `ex12.hs` wpisujemy

```
prodPrices p = case p of
 "A" -> 100
 "B" -> 500
 "C" -> 1000
 _   -> error "Unknown product"

products = ["A","B","C"]

-- basic discount strategy
discStr1 p
 | price > 999 = 0.3 * price
 | otherwise   = 0.1 * price
 where price = prodPrices p

-- flat discount strategy
discStr2 p = 0.2 * prodPrices p

totalDiscout discStr =
 foldl1 (+) .
 map discStr .
 filter (\p -> prodPrices p > 499)
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie

```
ghci> totalDiscout discStr1 ["A", "B", "C"]
ghci> totalDiscout discStr2 ["A", "B", "C"]
```

3. ***Zadania***:
    1. (*opcjonalne*) Poprawić (zrefaktoryzować) powyższy kod (aspekty: czytelności, ogólności, wydajności)

2. Obliczyć w GHCi i przeanalizować wyrażenia

```
ghci> replicate 2 . product . map (*3) $ zipWith max [4,2] [1,5]
ghci> sum . takeWhile (<1000) . filter odd . map (^2) $ [1..]
ghci> length . fromList . Prelude.map toLower $ "thirteen men must go" -- potrz
```